操作系统课程设计实验报告

*姓名：*

*学号：*

# 实验1 Kernel API

* 1. **实验目的**

通过本实验的学习，掌握信创操作系统内核定制中所常用的内核数据结构和函数，具体包括：内核链表、内核内存的分配和释放、内核线程、内核锁。

* 1. **实验内容**

设计一个内核模块，并在此内核模块中创建一个内核链表以及两个内核线程

线程1需要遍历进程链表并将各个进程的pid、进程名加入到内核链表中

线程2中需不断从内核链表中取出节点并打印该节点的元素

在卸载模块时停止内核线程并释放资源

* 1. **实验方法**
     1. **进入在线实验平台中的VM环境中以进行实验**
        1. 关键步骤：

使用课程提供的在线操作平台进行登录，创建Pod和VM。

先远程连接Pod，再在Pod中利用集群内网环境对VM进行连接。

* + - 1. 关键技术：

ssh（连接虚拟机），vi（编辑代码)

* + 1. **在VM环境中前置学习**
       1. 关键步骤：

Linux基本命令(ls, cd,cp, vi, make, gcc等)

内核模块概念，编程框架

内核线程创建编写以及销毁进行了解

内核锁的获取、使用以及释放进行了解

内核态的内存分配以及释放进行了解

* + - 1. 关键技术：

Deepseek和其他大模型，编小例子测试

* + 1. **代码编写**

在本地Windows系统中使用Vscode进行代码编写

将代码文件粘贴到VM中编译运行，进行调试完善

* 1. **实验结果**
     1. **加载模块输出结果**

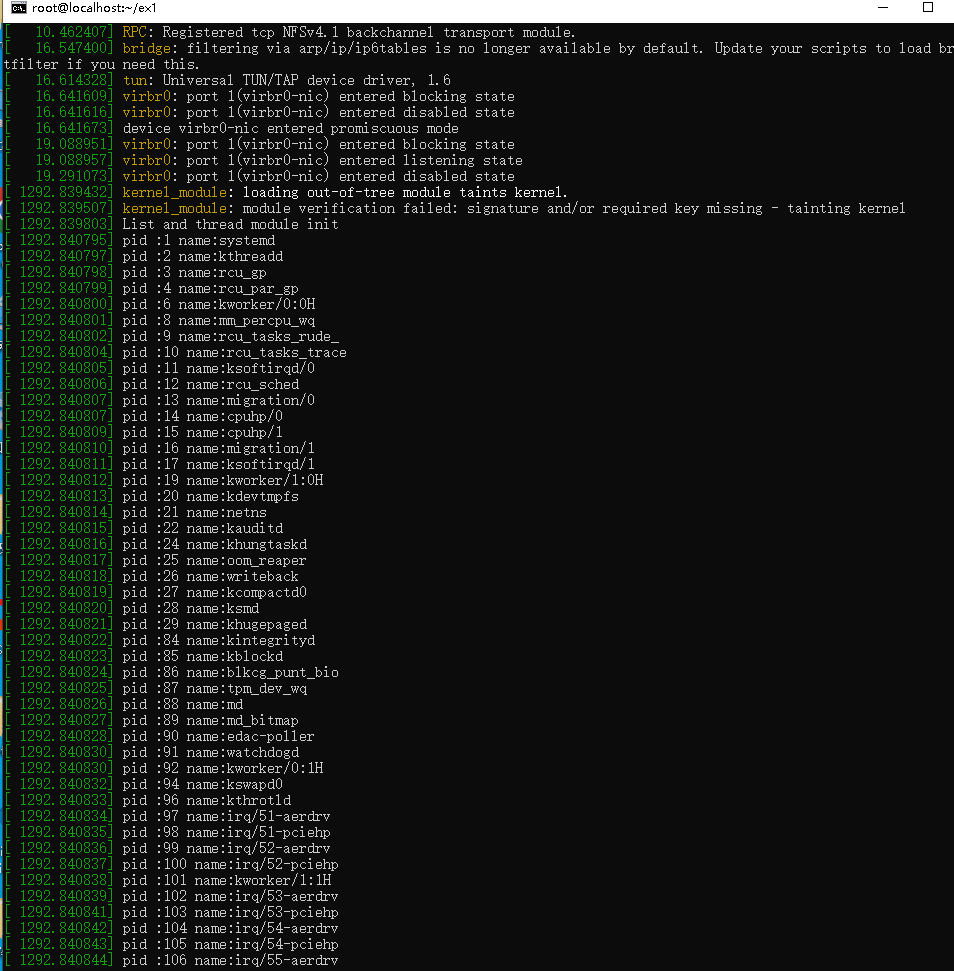


图 1.1加载模块输出结果果

* + 1. **卸载模块后输出结果**



图 1.2卸载模块后输出结果

* 1. **思考题**
     1. **kmalloc, kfree与C语言的malloc，free有何不同？**
        1. 内存分配空间不同​

kmalloc 和 kfree 是内核空间的内存管理函数，用于在内核态分配和释放内存。

malloc 和 free 是用户空间的库函数，用于在用户态分配和释放内存。

* + - 1. 内存分配特性​

kmalloc 分配的是物理连续的地址空间（受限于内核碎片情况），而 malloc 分配的是虚拟地址空间（可能不连续）。kmalloc 需要指定分配标志（如GFP\_KERNEL 或GFP\_ATOMIC），以控制是否允许休眠或原子操作；malloc 不需要此类参数。

* + - 1. 使用场景​

kmalloc 适用于内核模块、中断上下文等需要直接操作硬件的场景。

malloc 用于用户程序，无法访问内核资源。

* + - 1. 释放要求​

kfree 必须释放由 kmalloc 分配的内存，free 必须释放由 malloc 分配的内存，否则会导致未定义行为。

* + 1. **通过何种函数获得进程的PCB？**

在Linux内核中，进程的PCB由task\_struct结构体表示。

通过遍历进程链表（使用宏for\_each\_process(p)，其中p是遍历过程中指向每个进程 task\_struct 的指针）来获取当前进程（current 宏可直接获取当前执行线程的 task\_struct）。

* + 1. **两个内核线程如何实现同步互斥？**

通过自旋锁spinlock实现同步互斥。线程1在操作前调用spin\_lock(&lock)，操作后调用spin\_unlock(&lock)。线程2（从链表取出节点）同样在操作前后加锁和解锁。

* + 1. **内核线程和用户空间线程有何不同？**
       1. 运行环境

内核线程运行在内核态，共享内核地址空间运行在用户态，用户空间线程属于某个进程的地址空间。

* + - 1. 调度方式

内核线程由内核直接调度，用户空间线程由用户线程库（如pthread）或内核调度。

* + - 1. 资源访问

内核线程可访问内核数据结构和硬件资源，用户空间线程仅能通过系统调用访问内核资源。

* + - 1. 创建方式

内核线程通过kthread\_create创建，需内核权限，用户空间线程通过pthread\_create创建。

* + - 1. 典型用途

内核线程处理中断、磁盘缓存、网络协议栈等后台任务应用程序的多任务处理。

* + 1. **使用内核链表而不是自行编写链表的好处？**
       1. 标准化与可靠性

内核链表经过严格测试，避免手动实现可能引入的错误（如指针操作错误）。

* + - 1. 灵活性

链表节点仅包含指针域，可嵌入任意结构体中，例如实验中的 pid\_node。

* + - 1. 丰富的API支持

提供list\_add\_tail、list\_del、list\_for\_each\_safe等宏和函数，简化链表操作。

* + - 1. 性能优化

内核链表为双向循环链表，插入、删除操作时间复杂度为O(1)，效率高。

* + - 1. 代码复用

避免重复造轮子，减少代码冗余，提升可维护性。

# 实验2 deferred work

* 1. **实验目的**

通过本实验的学习，掌握信创操作系统内核定制中所常用的workqueue的使用，理解其与kernel thread的区别；

* 1. **实验内容**

分别用work queue和kernel thread两种方式调用10个函数（函数内按学号后3位加1区分，如315→316→317…），观察执行顺序差异。每个函数对应一个独立的kernel thread（共10个不同线程）。

探究work queue的delayed\_work功能，在模块加载5秒后打印预设信息以验证延迟执行效果

* 1. **实验方法**
     1. **Work Queue理论学习**
        1. WorkQueue的由来

WorkQueue由Bottom-Half机制演化而来。

* + - 1. WorkQueue的概念

一个函数队列中，由单个线程遍历队列顺序执行其中的函数，这个队列就是WorkQueue。

* + - 1. WorkQueue与Kernel Thread的理论区别

WorkQueue顺序执行任务，而Kernel Thread并发执行任务。

* + 1. **代码编写：**
       1. WorkQueue初始化方式

INIT\_WORK初始化work\_ctx（需包含struct work\_struct作为首参数）。

work\_queue\_handler为任务处理函数，需通过container\_of从work\_ctx提取current\_id。

* + - 1. WorkQueue生成方法

使用schedule\_work加入系统默认WorkQueue。

* 1. **实验结果**

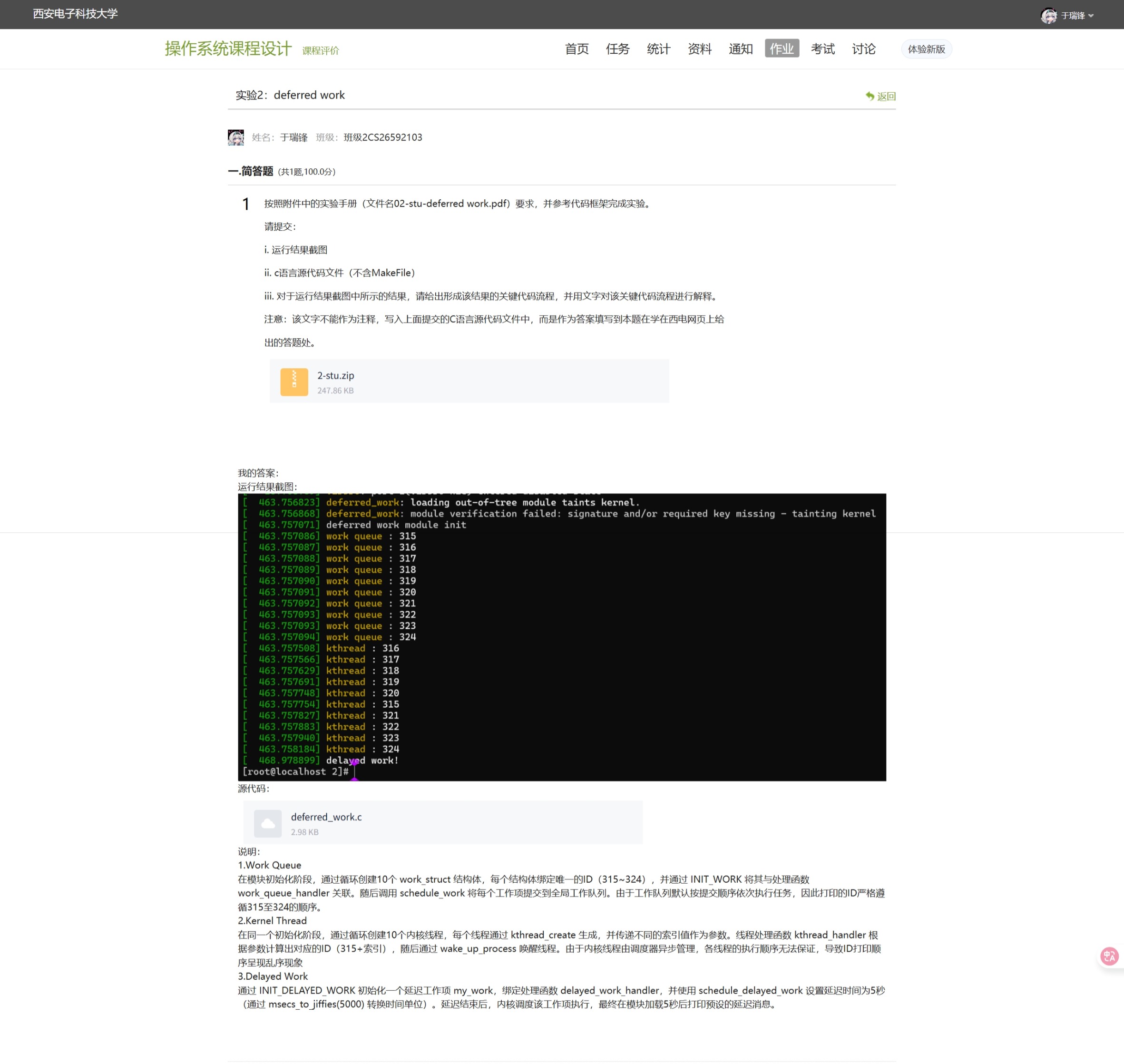


图 2.1work queue和kernel thread的执行结果

可以看出，workqueue按顺序执行，而kernel thread的执行无序。

* 1. **思考题**

基于实验结果，解释workqueue与kernel thread的不同。

* + 1. **任务调度机制不同**

Workqueue通过 schedule\_work() 将任务提交到全局工作队列后，任务按提交顺序依次执行。即使存在多个工作项，内核默认的全局工作队列会以单线程方式逐个处理任务，形成严格的FIFO（先进先出）顺序。因此实验中观察到 ID 严格递增。

Kernel Thread每个线程通过 wake\_up\_process() 独立唤醒后，由内核调度器异步管理执行顺序。线程之间竞争 CPU 资源，其执行顺序受系统负载、中断、优先级等因素影响，呈现非确定性乱序。实验中线程ID 打印顺序验证了这一点。

* + 1. **并发控制粒度不同**

Workqueue任务执行单元是工作项，多个工作项共享同一内核线程。任务间可能因共享执行线程而产生隐式同步，但开发者无需显式管理线程生命周期。

Kernel Thread每个任务对应一个独立的内核线程，线程与任务一一绑定。线程的创建、调度、销毁需开发者通过kthread\_create() 和kthread\_stop() 显式管理。

# 实验3 Edu驱动

* 1. **实验目的**

通过本实验的学习，掌握信创操作系统内核定制中所常见PCI设备驱动适配技术。

* 1. **实验内容**

补全框架中的TODO位置的缺失代码。

实现驱动程序的ioctl调用处理功能。该调用需接收一个整型参数。当驱动程序接收到用户的ioctl调用后，需创建一个内核线程。在该内核线程中，利用edu设备的阶乘功能对传入的整型参数进行计算，并将计算结果存储于驱动程序中，以便用户进程后续获取。驱动程序需具备识别不同进程调用的能力，确保将计算结果正确返回给对应的调用进程。

编写 C 语言应用程序，通过调用edu驱动的ioctl接口进行操作。首先，设置参cmd值为0，输入待计算的数值。等待一定时间后，将参数cmd值更改为1，再次调ioctl接口，以获取设备计算完成的结果。

* 1. **实验方法**
     1. **PC驱动理论知识学习**

驱动工作流程的核心目标，PCI（Peripheral Component Interconnect）驱动是操作系统与PCI硬件设备之间的桥梁，其核心任务是实现以下两点：

* + - 1. **与操作系统对接**

这是驱动开发者需要遵循的“软件规则”，具体流程包括：

* + - * 1. 驱动注册与绑定

驱动模型：在操作系统（如Linux）中，PCI驱动需注册为 pci\_driver 结构体，声明支持的设备ID列表（pci\_device\_id）。

设备发现：操作系统通过PCI总线枚举设备时，匹配驱动与设备的ID，触发驱动的 probe() 函数（如用户实验中的 edu\_driver\_probe）。

* + - * 1. 资源管理

内存与中断分配：操作系统为设备分配内存空间（如映射BAR地址到虚拟内存）、中断号（IRQ）等资源。

接口暴露：驱动需向用户态提供标准接口（如字符设备文件 /dev/edu），并通过 ioctl、read/write 等系统调用与应用程序交互（对应实验中的 edu\_dev\_unlocked\_ioctl）。

* + - * 1. 生命周期管理

加载与卸载：驱动模块加载时初始化，卸载时释放资源（如实验中的 edu\_driver\_remove 需反向操作 probe() 的步骤）。

热插拔支持：动态响应设备的插入或移除事件。

* + - 1. **与PCI设备对接**

这是驱动开发者需要实现的“硬件操作”，具体内容包括：

* + - * 1. PCI配置空间访问

设备识别：通过PCI配置空间读取设备ID、厂商ID、BAR寄存器信息等。

功能配置：设置设备的工作模式（如启用DMA、中断类型）。

* + - * 1. BAR寄存器映射

物理地址转换：将BAR中的总线地址映射到系统虚拟地址（如实验中的 ioremap），使驱动能通过内存读写操作控制设备。

设备寄存器操作：通过映射后的虚拟地址访问设备寄存器（如触发计算、读取状态）。

* + - * 1. 中断与数据传输

中断处理：注册中断处理函数，响应设备完成操作的通知（如阶乘计算完成）。

DMA与内存同步：若设备支持DMA，需管理缓存一致性和数据传输（实验中未涉及，但属于通用流程）。

* + 1. **代码编写**
       1. **初始化流程**​

模块加载时触发edu\_driver\_init的执行，完成两项任务：

* + - * 1. 字符设备文件操作注册

目的：允许用户程序通过字符设备文件（如/dev/edu）访问设备。

* + - * 1. PCI设备操作注册

目的：使Linux系统能初始化PCI驱动。

设备发现机制：PCI驱动注册成功后，内核自动调用edu\_driver\_probe函数，负责初始化匹配的PCI设备。edu\_driver\_probe功能：为edu设备申请资源（如地址空间、IRQ号），并将资源状态记录到edu\_dev\_info结构体中。关键函数：pci\_ioremap\_bar：将PCI设备的BAR（基址寄存器）中的总线地址映射为内核可用的虚拟地址。

* + - 1. **用户程序操作流程**​
         1. 通过open打开 /dev/edu 设备时

触发 edu\_dev\_open 函数执行。分配内存userdata（用于存储阶乘计算结果）。将该内存与文件句柄（struct file\*）关联（通过private\_data字段）。

* + - * 1. 通过ioctl操作/dev/edu文件句柄时

触发edu\_dev\_unlocked\_ioctl函数执行。根据用户传入的cmd和arg参数决定启动内核线程或者读取结果。

启动内核线程：向edu设备寄存器（偏移量 0x08）写入 arg 值，触发阶乘计算。计算结果存入userdata。

读取结果：从userdata中获取阶乘值。

* + - * 1. 通过close关闭文件句柄时

触发edu\_dev\_release函数执行，释放userdata内存。

* + - 1. 模块卸载流程​

模块卸载时触发 edu\_driver\_exit 执行，完成字符设备文件操作注销和PCI设备操作注销。触发edu\_driver\_remove函数自动调用，需完成取消地址映射（通过iounmap释放虚拟地址映射）和释放资源（销毁edu\_dev\_info结构体占用的内存）。

* + - 1. Makefile 编译问题

Makefile中编译目标为edu\_dev.c，但提供的源码文件名为edu\_dev\_blank.c。需手动修改文件名或调整Makefile配置，否则编译失败。

* 1. **实验结果**

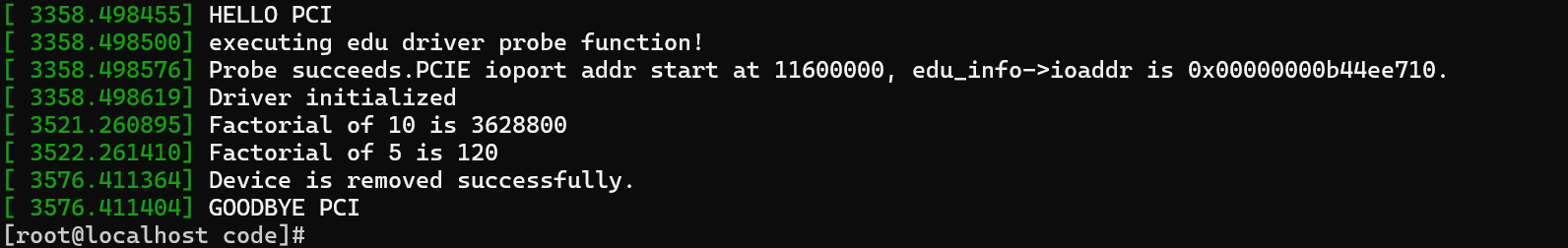


图 3.1edu驱动的执行结果

可以看到程序正常运行，使用了设备。

* 1. **思考题**
     1. **解释所填写的实验代码的语义**
        1. PCI设备初始化（edu\_driver\_probe）​

内存分配：通过kzalloc为设备信息结构体edu\_dev\_info分配内存，保存设备资源信息。

资源请求：调用pci\_request\_regions保留PCI设备I/O和内存资源，防止资源冲突。

地址映射：使用pci\_ioremap\_bar将设备的BAR0寄存器映射到内核虚拟地址空间，后续可通过edu\_info->ioaddr访问设备寄存器。

私有数据绑定：通过pci\_set\_drvdata将设备结构体edu\_dev\_info关联到PCI设备，便于后续通过设备指针访问私有数据。

* + - 1. PCI设备卸载（edu\_driver\_remove）

资源释放：调用iounmap取消地址映射，pci\_release\_regions释放资源，kfree释放设备信息结构内存，完成清理。

* + - 1. 设备打开与关闭（edu\_dev\_open 和 edu\_dev\_release）

上下文绑定：在edu\_dev\_open中分配user\_data结构体，存储进程上下文（如进程ID），并绑定到filp->private\_data。

资源释放：在edu\_dev\_release中释放user\_data内存，避免内存泄漏。

* + - 1. Ioctl处理（edu\_dev\_unlocked\_ioctl）

命令分发：根据 cmd 值区分操作：

写入数据（cmd=0）：创建thread\_data保存用户输入值，启动内核线程kthread\_handler处理计算。

读取结果（cmd=1）：通过原子操作atomic64\_read安全返回计算结果。

线程管理：使用kthread\_create创建内核线程，wake\_up\_process启动线程，实现异步计算。

* + - 1. 内核线程（kthread\_handler）

阶乘计算：循环计算输入值的阶乘（此处为简化实现，实际应通过设备寄存器触发硬件计算）。

原子写入：使用atomic64\_set将结果写入user\_data->data，确保多进程访问的原子性。

内存释放：线程结束后释放thread\_data内存。

* + 1. **基于实验代码，请解释如何访问PCI设备的寄存器**

在驱动中，通过以下步骤访问PCI设备寄存器：

* + - 1. 地址映射

在edu\_driver\_probe中，使用pci\_ioremap\_bar(dev, 0)将设备的BAR0寄存器映射到内核虚拟地址edu\_info->ioaddr。

* + - 1. 寄存器读写

写入寄存器：使用writel(value, edu\_info->ioaddr + offset)向指定偏移量（offset）的寄存器写入数据。

读取寄存器：使用readl(edu\_info->ioaddr + offset)从寄存器读取数据。

* + - 1. 同步与等待

若设备计算需耗时，可通过轮询状态寄存器或中断机制等待设备完成操作。

* + 1. **请解释本驱动中启用线程的好处以及带来的问题**
       1. 好处

非阻塞操作：用户进程调用ioctl写入数据后无需等待计算完成，可继续执行其他任务，提高响应性。

并发支持：多个进程可同时发起计算请求，内核线程并行处理，充分利用多核CPU。

资源隔离：每个线程独立处理计算任务，避免单个进程长时间占用CPU。

* + - 1. 问题​

同步开销：需使用锁（如自旋锁）或原子变量确保共享数据（如计算结果）的访问安全，增加代码复杂度。

资源泄漏风险：若线程未正确释放内存（如 thread\_data），可能导致内存泄漏。

调度延迟：内核线程的调度依赖系统负载，可能导致计算延迟不确定。

竞态条件：多线程访问设备寄存器时需严格同步，否则可能引发硬件状态错误。

# 实验4 内核裁剪

* 1. **实验目的**

通过本实验的学习，掌握信创操作系统内核裁剪、根文件系统定制以及内核调参技术。

* 1. **实验内容**

利用qemu模拟器启动并运行一个虚拟机，以下是详细步骤：

修改Linux内核默认编译配置，移除对ext4文件系统的默认支持，并新增对btrfs文件系统的支持，随后进行内核编译。

利用buildroot工具构建根文件系统(rootfs)，配置Target options为最终运行该系统的平台类型。配置Filesystem images为btrfs格式，此操作将生成btrfs类型的虚拟磁盘，内含rootfs。配置Target packages，向rootfs中添加vim和bash。配置System configuration，将默认shell设置为bash。

通过qemu，添加共享文件夹和edu设备，结合已编译的内核及虚拟磁盘，尝试启动并运行基于btrfs虚拟磁盘的虚拟机。

将在qemu虚拟机外编译好的edu驱动程序复制到虚拟机内，并进行安装和运行测试。

使用默认配置的内核，进入虚拟机后查询内核参数kernel.shmmax的值，并尝试修改该参数。修改完毕后，再次查询kernel.shmmax的值，以验证修改是否成功。

* 1. **实验方法**
     1. **内核定制、rootFS定制和内核与rootFS对接之间的关系概述**
        1. 内核定制与rootFS定制的依赖关系

内核定制需要支持rootFS的文件系统类型。实验中要求内核移除ext4支持并添加btrfs支持，而rootFS被配置为btrfs格式。若内核未启用btrfs驱动，即使rootFS构建成功，内核也无法挂载该文件系统，导致虚拟机启动失败。

内核功能直接影响rootFS的运行环境。例如，内核需提供必要的硬件驱动（如edu设备驱动）、系统调用支持，以及用户空间工具（如bash、vim）所需的底层接口。

* + - 1. rootFS定制对内核功能的补充

rootFS提供用户空间工具和服务，依赖内核支持其运行。

bash作为默认shell需要内核的终端（TTY）和进程管理功能。

sysctl工具（用于修改kernel.shmmax参数）依赖内核暴露的/proc/sys接口。

rootFS的初始化配置需与内核参数匹配。例如，内核启动参数root=/dev/vda需指向正确的rootFS设备路径，且rootFS中需包含初始化脚本（如init）以启动用户空间进程。

* + - 1. 内核与rootFS对接的实现

启动参数与设备配置是两者对接的关键。QEMU命令中通过-kernel指定编译后的内核，-driver指定rootFS镜像，-append传递内核参数（如root=...），确保内核能正确加载rootFS。

内核需在启动时识别rootFS的设备（如虚拟磁盘），并调用对应的文件系统驱动（如btrfs）完成挂载。若驱动缺失或配置错误，系统将无法启动。

* + 1. **内核定制的实现**
       1. 内核编译​

运行make menuconfig 进入内核配置界面。

导航至File systems子菜单，取消勾选ext4文件系统支持，并勾选btrfs文件系统支持。保存配置并退出。

执行make -j$(nproc) Image.gz，生成压缩内核镜像Image.gz。

* + - 1. RootFS系统编译​

使用buildroot工具配置根文件系统。运行make menuconfig进入配置界面。Filesystem images选择btrfs格式。Target packages添加vim和bash。System configuration设置默认Shell为bash。

执行make -j$(nproc)生成文件系统镜像。

* + - 1. 内核与RootFS对接

QEMU启动命令放在了实验文件夹的一个shell脚本下，chmod +x添加执行权限之后运行脚本即可启动虚拟机。

在虚拟机内执行：mkdir /root/qemu-share和mount -t 9p -o trans=virtio host\_share /root/qemu-share，将宿主机文件复制到共享目录后，虚拟机内可加载edu驱动和测试程序。

* 1. **实验结果**

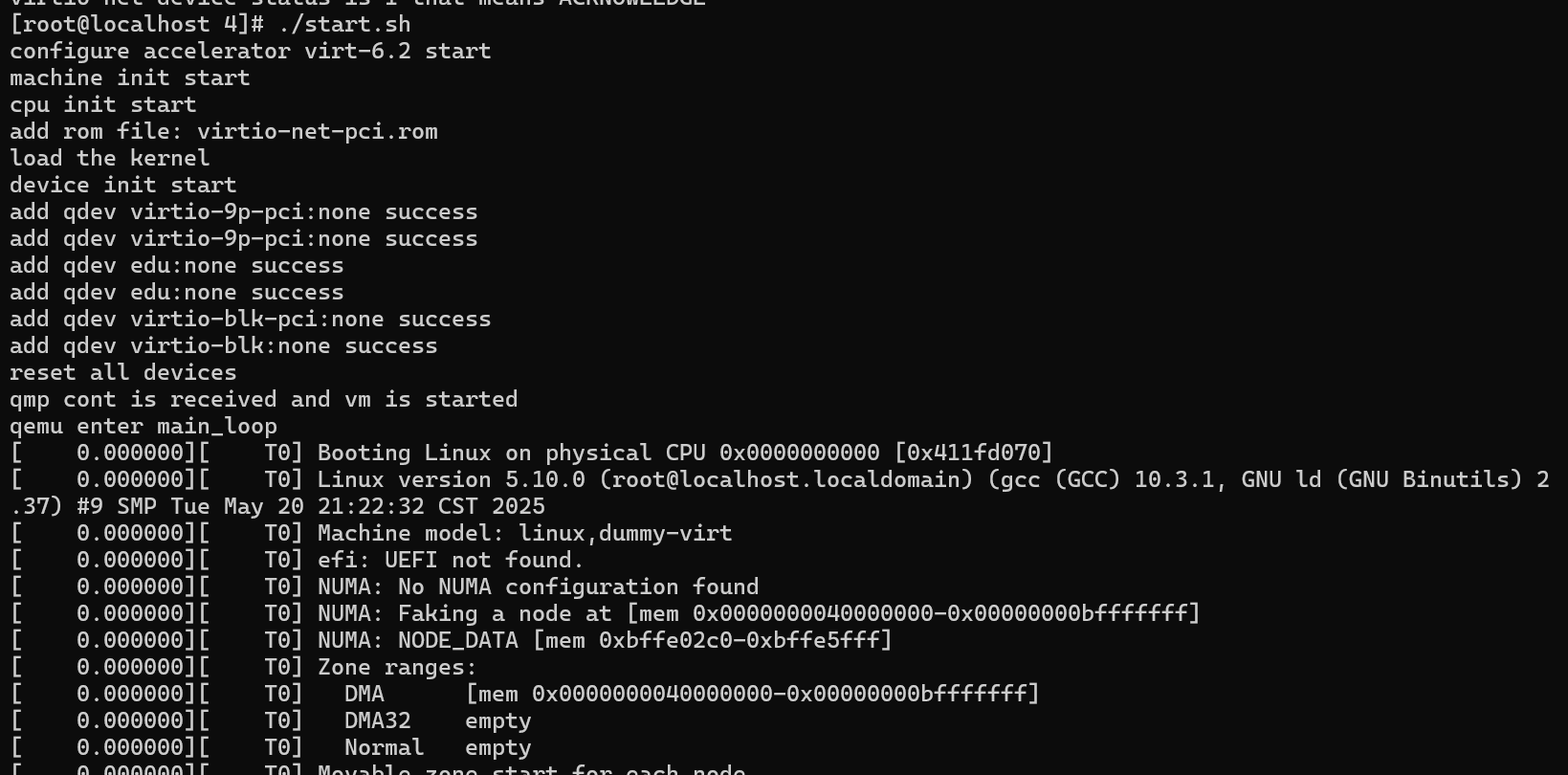


图 4.1qemu启动虚拟机

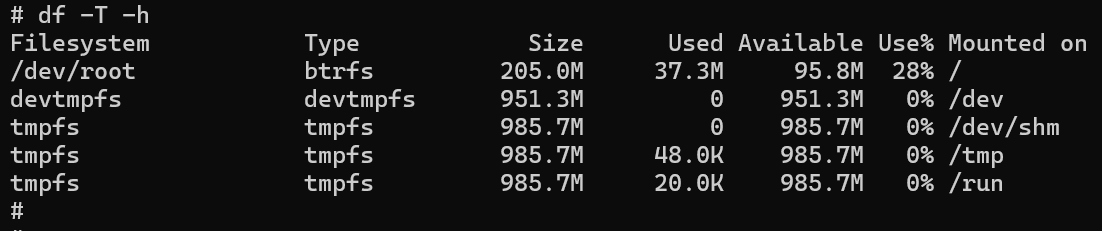


图 4.2查看定制系统文件系统类型

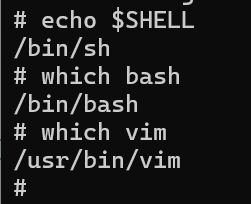


图 4.3查看定制系统vim和bash

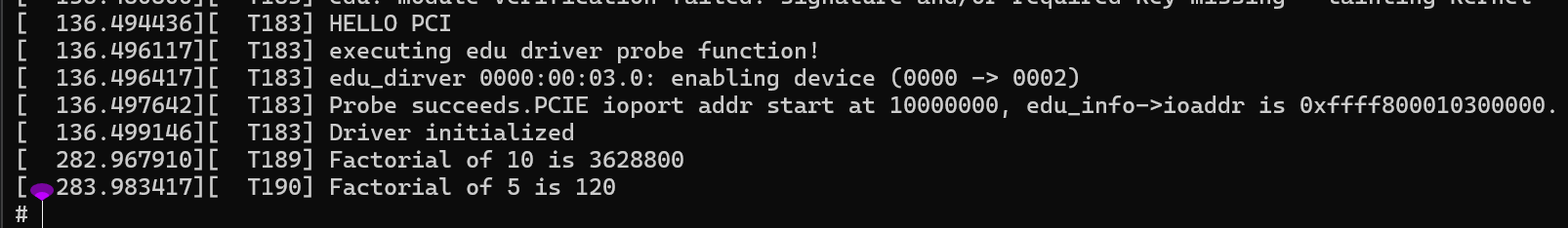


图 4.4定制系统中edu驱动的运行结果

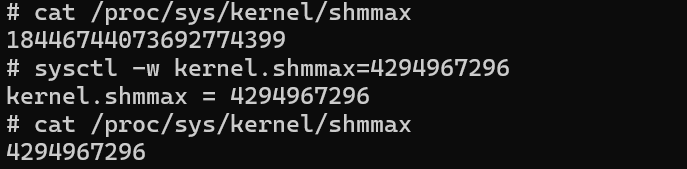


图 4.5修改内核参数

* 1. **思考题**
     1. **编译内核的关键步骤及解释**
        1. 关键步骤
           1. 配置内核选项

执行make menuconfig进入内核配置界面。

导航至File systems子菜单，取消勾选ext4支持，并勾选btrfs支持。保存配置后退出。

* + - * 1. 编译内核

使用 make -j$(nproc) Image.gz 命令，生成压缩内核镜像 Image.gz。

* + - 1. 解释

make menuconfig通过文本界面（TUI）修改内核配置，移除默认的ext4文件系统支持并新增btrfs文件系统，确保虚拟机后续能正确识别 btrfs 格式的根文件系统。

-j$(nproc) 表示使用与 CPU 核心数相同的并行编译任务，加速编译过程。

* + 1. **编译buildroot的关键步骤及解释**
       1. 关键步骤
          1. 配置 Buildroot 选项

执行make menuconfig进入配置界面。

Target options设置为目标平台类型。

Filesystem images选择btrfs格式。

Target packages添加vim和bash。

System configuration将默认Shell设置为bash。

* + - * 1. 构建根文件系统

执行make -j$(nproc)编译生成 btrfs 格式的根文件系统镜像。

* + - 1. 解释

Target options 确保生成的根文件系统与目标平台指令集兼容。

Filesystem images选择btrfs是为了匹配内核配置，使虚拟机可挂载该格式的根文件系统。

添加vim和bash是为了提供必要的工具支持。

* + 1. **如何将edu驱动文件放入根文件系统中的过程**

共享文件夹方式

* + - 1. Host 侧准备

将编译好的驱动文件复制到Host机的/root/share目录。

* + - 1. 虚拟机内挂载共享文件夹

在虚拟机内创建目录：mkdir /root/qemu-share。

* + - 1. 挂载Host目录

mount -t 9p -o trans=virtio,version=9p2000.L [mount\_tag] /root/qemu-share（需替换 [mount\_tag] 为实际标签）。

* + - 1. 复制驱动文件

从/root/qemu-share复制驱动文件到虚拟机内的目标位置。

* + 1. **修改kernel.shmmax的值的应用场景**

kernel.shmmax定义单个共享内存段的最大大小，修改该参数的典型场景包括：

* + - 1. 数据库优化

如PostgreSQL或Oracle使用大量共享内存缓存数据，需调高shmmax以避免内存不足。

* + - 1. 高性能计算（HPC）

多进程应用（如MPI程序）依赖共享内存通信时，需增大该值以支持更大数据块交换。

* + - 1. 实时数据处理

金融交易系统或流处理框架需快速共享数据，合理设置shmmax可提升性能。

* + 1. **在x86平台上编译arm平台下所使用的根文件系统，是否可行**
       1. 可行性

可行。

* + - 1. 理由
         1. 交叉编译支持

Buildroot设计上支持交叉编译，只需在Target options中指定目标平台为ARM。

* + - * 1. 工具链配置

选择预编译的ARM交叉编译工具链（如gcc-arm-linux-gnueabihf）或使用Buildroot自动下载并构建工具链。

* + - * 1. 镜像兼容性

生成的根文件系统镜像格式（如btrfs）与平台无关，仅需确保内核针对ARM平台编译并正确配置。

# 个人体会

* 1. **遇到的关键问题及解决方法**
     1. **实验1内核链表同步问题**

在实现双线程操作内核链表时，首次运行发现打印结果存在重复或丢失。通过以下步骤定位：

在dmesg日志中发现"list\_del corruption"错误提示。

用printk跟踪链表操作流程，发现线程2取出节点后未及时加锁。

解决方法：在链表插入/删除操作前后增加自旋锁保护，并改用list\_for\_each\_entry\_safe安全遍历宏。

* + 1. **实验3 edu驱动问题​**

执行用户测试程序一直检测不到设备，经查阅资料后发现要先用命令添加设备节点并赋予权限。

执行mknod /dev/edu c 200 0和chmod 666 /dev/edu命令后成功。

* 1. **启发与收获**
     1. **技术层面**

主要是系统性调试能力，比如掌握dmesg日志分析printk打印中间结果等调试方法。

* + 1. **学习方法**

分步验证策略

文档优先原则

模块化思维